SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

November 2014, Vol. 57 112110:1–112110:19
doi: 10.1007/s11432-014-5168-7

Automated enforcement for relaxed information release with reference points

SUN Cong[1]*, XI Ning[1], GAO Sheng[1], CHEN Zhong[2] & MA JianFeng[1]

[1]School of Computer Science and Technology, Xidian University, Xi'an 710071, China;
[2]School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China

Received March 15, 2014; accepted June 9, 2014; published online September 3, 2014

Abstract Language-based information flow security is a promising approach for enforcement of strong security and protection of the data confidentiality for the end-to-end communications. Here, noninterference is the standard and most restricted security property that completely forbids confidential data from being released to public context. Although this baseline property has been extensively enforced in various cases, there are still many programs, which are considered secure enough, violating this property in some way. In order to control the information release in these programs, the predetermined ways should be specified by means of which confidential data can be released. These intentional releases, also called declassifications, are regulated by several more relaxed security properties than noninterference. The security properties for controlled declassification have been developed on different dimensions with declassification goals. However, the mechanisms used to enforce these properties are still unaccommodating, unspecific, and insufficiently studied. In this work, a new security property, the Relaxed Release with Reference Points (R3P), is presented to limit the information that can be declassified in a program. Moreover, a new mechanism using reachability analysis has been proposed for the pushdown system to enforce R3P on programs. In order to show R3P is competent for use, it has been proved that it complies with the well-known prudent principles of declassification, and in addition finds some restrictions on our security policy. The widespread usage, precision, efficiency, and the influencing factors of our enforcement have been evaluated.

Keywords information flow, security policy, noninterference, declassification, pushdown system, program analysis

Citation Sun C, Xi N, Gao S, et al. Automated enforcement for relaxed information release with reference points. Sci China Inf Sci, 2014, 57: 112110(19), doi: 10.1007/s11432-014-5168-7

1 Introduction

It has been long since the research community has been evaluating the solutions for preventing the confidential information in computing system from being improperly leaked for unauthorized access. Cryptographic efforts mainly provide computational infeasibility which prevent the plaintext or key from being calculated. Access control mechanisms also provide fine-grained approaches that avoid the invalid access to confidential data. Although both approaches can forbid confidential data to be directly exposed to public, neither of them can prevent programs that have been authorized to access confidential data from propagating the confidential information to unauthorized party. This kind of unintended leakage often

*Corresponding author (email: suncong@xidian.edu.cn)

© Science China Press and Springer-Verlag Berlin Heidelberg 2014 info.scichina.com link.springer.com

relies on the public observable channels and other constructs (e.g., variables, object fields, exceptions) of program. The problem has drawn increasing attention from the communities of security and programming language, and is recognized as information flow security [1,2].

The ideal objective of information flow security, referred to as noninterference [3], is to prevent any possible leakage and to ensure the system to be completely secured. This objective is often too unaccommodating for real time usage. For example, consider a scenario of simple authentication,

$$\text{if (guess == password[id]) then} /* \text{pass} */ \text{else} /* \text{fail} */;$$

this program compares the public message guess with the confidential password of some *id* and releases a single bit decision on whether this message is correct or not. This scenario violates noninterference. In fact, this kind of deliberate release is legitimate. Moreover, allowing intentional information release drives us to clearly specify the target of release, e.g., the difference between the above guess==password[id] and guess==password[id]%10000. Thus, it is critical to regulate the safe release. Therefore, it becomes indispensable to develop more relaxed security policies and the corresponding enforcement mechanisms for the intentional information releases [4]. The more relaxed and practical policies, known as declassification policies, specify the context under which the confidential information is permitted to be released. The declassifications can be categorized with different intentions along four dimensions [5], respectively capturing what information is released, where in the system does the release happen, when the information can be released and who releases information. Beyond the specification of information flow security policies and the related security properties of program, an even more crucial issue is to develop enforcement for each policy. The related techniques to enforce this kind of security properties include type systems, program logics, abstract interpretation, automated verification, program slicing based on dependent graphs, and runtime monitoring [6].

In this work, we propose Relaxed Release with Reference Points (R3P) as a security property on the what-dimension of declassification. Similarly to the security property WERP [7], this property resorts to the explicitly specified reference points to locate the positions where deliberately released expressions become declassifiable. We observe the differences between R3P and the most prevalent security properties. Our security property is more general than the previous properties that are enforced through automated verification [8,9]. To get a better understanding of our security property, we evaluate the new security property against existing prudent principles [5,10]. From the evaluations we also develop some new principles and find the auxiliary conditions to make R3P comply with several principles.

For the enforcement, we give an automatic approach based on reachability analysis of pushdown system. We use the principle of reachability analysis first reported in our previous work [11]. The reachability analysis can ease the verification effort by avoiding temporal logic formula specification or partial correctness assertions as used in [12,13]. This advantage relies on our improvement of the model transformation technique, self-composition [12], whose principle is to compose the model with a variable-renamed copy and reduce the security property of the original program to a safety property of the model after transformation. When the I/O channels are considered, a precise modeling of program requires each I/O to be modeled explicitly. Therefore an ordinary self-composition, which duplicates the I/O channels, will largely increase the state space of model. Considering this restriction, we improve the self-composition using a store-match pattern to reduce the state space of model. Moreover, we can exclude the irrelevant traces violating the precondition of the property by matching the pre-stored value of declassifiable expressions at specific reference points. The experimental results show the precision of our enforcement and explain several factors that have influence on the cost of reachability analysis.

Our main contribution is summarized as follows. First, we propose a security property more general than the previous ones, enforceable by automated verification. Second, we prove the compliance of our security property to several general prudent principles for declassifications, and find some non-trivial principle, i.e. conditional persistence, to figure out the restriction on the security policy. Third, we first use reachability analysis to enforce security property on the what-dimension of declassification. The enforcement is more general than the previous approach based on automated verification [8]. Moreover, we have also developed a novel technique, i.e. store-match pattern, for the model transformation to

reduce the state space and verification cost, in order to make our approach more scalable.

2 Related work

Information flow security has been considered to be crucial for the general system models. The tracking approach of information flow has been implemented to identify covert channels in secured operating system [14]. The automatic policy compliance has also been considered [15]. From a perspective of programming language, there is a tendency to develop more flexible security policies for controlling information releases on different dimensions of declassification [5], and to implement more scalable and practical mechanisms to enforce the properties related to these policies [4,6].

There have been several information flow security properties on regulating what information can be declassified by a program, e.g., delimited release [16], relaxed noninterference [17], WHAT_1, WHAT_2 [18], WERP [7], and the lattice-based property [19]. Sabelfeld and Myers [16] proposed delimited release, which first defines the escape hatch annotated with a primitive declassify to collect a series of expressions whose initial values are deliberately released through the escape hatches. Li and Zdancewic [17] gave an extensible framework of declassification policies. Their policy specifies how data can be declassified with λ-calculus term. The partial order relation on the sets of policies is the set inclusion of label interpretation. Terauchi and Aiken [8] extended relaxed noninterference from pure functional languages to imperative languages, and showed that by strengthening the specification of relaxed noninterference with semantic equivalence, the property is equivalent to delimited release. Mantel and Reinhard [18] adapted the principle of selective dependency [20] to a multi-thread setting to derive WHAT_1 and WHAT_2, where both rely on step-wise bisimulation with respect to a set of pairs of escape hatches. Adetoye and Badii [19] proposed a lattice model to capture the relative safe level of information releases. This novel use of lattice could unify a variety of representations of information. The policy is mostly abstract, and no enforcement was presented explicitly. These security properties given above on the what-dimension of declassification only permit deliberate release of the initial values of declassifiable expressions. When we need to release the intermediate values of declassifiable expressions, we cannot specify the security condition exactly with these properties. Lux and Mantel [7] proposed the security property WERP, incorporating specific hatch validation function and invalidation function to stipulate the confidential data at specific reference points to be declassified. With WERP and the R3P presented in this work, we can release the intermediate values of declassifiable expressions depending on the position of reference points.

Several other security properties extend the consideration on which information can be released with a consideration on where in the program the declassification can happen. The bisimulation-based security property, localized delimited release [21], takes into account the value of expressions from a pair of expression set at each step. The intermediate values of expressions are used to decide whether the stepwise release of expression is under control. The recent security property, WHAT&WHERE, proposed by Lux et al. [22] is based on a declassification policy for a concurrent language with scheduler and dynamic thread creation. The authors developed both scheduler-specific and scheduler-independent version of security properties using local escape hatches to constrain in places where certain secrets can be released.

From a perspective of enforcement mechanisms, although type system is pervasively accepted to enforce properties on each dimension of declassification [7,16–18,21,22], the approaches are mostly theoretical. And the well-implemented compiler, Jif [1)], only supports the who-dimensional declassification via decentralized label model [23]. Compared with the type-based approaches, automated verification is considered more precise and has been adopted to enforce noninterference. The target system can vary from multilevel security systems [24] to the programs developed with imperative languages [8,9,12] or object-oriented languages [11,13]. In these work, the only ones discussing enforcement of declassification are [8,9]. Terauchi and Aiken leveraged a model checker to enforce the what-dimensional relaxed noninterference [8]. We proposed an enforcement using reachability analysis for a where-dimensional security property, where-security [9]. This is the first attempt to apply reachability analysis on declassification policy enforcement.

1) http://www.cs.cornell.edu/jif.

$$e ::= v \mid x \mid e\ op\ e' \qquad C ::= C;C' \mid r : c$$
$$c ::= \textbf{skip} \mid x := e \mid x := declass(e,r) \mid \textbf{if}\ e\ \textbf{then}\ C\ \textbf{else}\ C' \mid \textbf{while}\ e\ \textbf{do}\ C \mid input(x, I_i) \mid output(e, O_i)$$

Figure 1 Program syntax.

$$\frac{}{(\mu,\mathcal{I},\mathcal{O},p,q,\textbf{skip};C) \to (\mu,\mathcal{I},\mathcal{O},p,q,C)} \qquad \frac{\mu(e)=v}{(\mu,\mathcal{I},\mathcal{O},p,q,x:=e;C) \to (\mu[x\mapsto v],\mathcal{I},\mathcal{O},p,q,C)}$$

$$\frac{\mu(e)=b}{(\mu,\mathcal{I},\mathcal{O},p,q,\textbf{if}\ e\ \textbf{then}\ C_{\textbf{true}}\ \textbf{else}\ C_{\textbf{false}}) \to (\mu,\mathcal{I},\mathcal{O},p,q,C_b)} \qquad \frac{(\mu,\mathcal{I},\mathcal{O},p,q,C_1) \to (\mu',\mathcal{I}',\mathcal{O}',p',q',C_1')}{(\mu,\mathcal{I},\mathcal{O},p,q,C_1;C_2) \to (\mu',\mathcal{I}',\mathcal{O}',p',q',C_1';C_2)}$$

$$\frac{\mu(e)=\textbf{true}}{(\mu,\mathcal{I},\mathcal{O},p,q,\textbf{while}\ e\ \textbf{do}\ C) \to (\mu,\mathcal{I},\mathcal{O},p,q,C;\textbf{while}\ e\ \textbf{do}\ C)} \qquad \frac{\mu(e)=\textbf{false}}{(\mu,\mathcal{I},\mathcal{O},p,q,\textbf{while}\ e\ \textbf{do}\ C) \to (\mu,\mathcal{I},\mathcal{O},p,q,\textbf{skip})}$$

$$\frac{I_i[p_i]=v \qquad p_i'=p_i+1}{(\mu,\mathcal{I},\mathcal{O},p,q,input(x,I_i);C) \to (\mu[x\mapsto v],\mathcal{I},\mathcal{O},p',q,C)} \qquad \frac{\mu(e)=O_i'[q_i] \qquad q_i'=q_i+1}{(\mu,\mathcal{I},\mathcal{O},p,q,output(e,O_i);C) \to (\mu,\mathcal{I},\mathcal{O}',p,q',C)}$$

$$\frac{\mu(e)=v}{(\mu,\mathcal{I},\mathcal{O},p,q,x:=declass(e,r);C) \to (\mu[x\mapsto v],\mathcal{I},\mathcal{O},p,q,C)}$$

Figure 2 Operational semantics.

In this work, we extend this approach to the enforcement of what-dimensional security property.

3 Language model

Deterministic imperative language is employed as the presentation language. The syntax includes both the I/O operations and a declassification primitive, see Figure 1. e and C are the syntax of expressions and commands respectively. x denotes a variable, and v stands for a constant value. Let V be the set of variables. We have $x \in V$. op is a meta-level binary operator on expressions. A program is a sequence of commands. Each command consists of a reference label r and an instruction c. $\mathcal{I} = \{I_i \mid i \in \mathbb{N}\}$ is the set of input channels. $\mathcal{O} = \{O_i \mid i \in \mathbb{N}\}$ is the set of output channels. $input(x, I_i)$ means that variable x obtains an input from I_i, and $output(e, O_i)$ stores the value of expression e to O_i. The instruction *declass* supplements the ordinary assignment with the information downgrading on the value of expression.

The operational semantics of program are given in Figure 2. We define the inductive rules over configurations. Each configuration is in the form of $(\mu, \mathcal{I}, \mathcal{O}, p, q, C)$. In a configuration, $\mu : V \mapsto \mathbb{N}$ is a memory store mapping each variable to a value. C is the sequence of commands to be executed. $\mu(e)$ means the evaluation of e under μ. $\mu[x \mapsto v]$ means changing μ by mapping variable x to a value v. The reference labels are treated syntactically for the security policy and security property, however they are irrelevant to the semantics. $p = \{p_i \mid i \in \mathbb{N}\}$ and $q = \{q_i \mid i \in \mathbb{N}\}$ are sets of indexes. p_i is the index of next element to be input from I_i. q_i denotes the index of location of O_i where the next output value will be stored in. The indexes of channels are explicitly increased by each step of the input and output computation. The channels are identical if they have the same length and identical contents. For any I_i, I_j, we define $I_i = I_j$ as $(p_i = p_j) \wedge \bigwedge_{0 \leqslant k < p_i}(I_i[k] = I_j[k])$. Similarly, $O_i = O_j$ means $(q_i = q_j) \wedge \bigwedge_{0 \leqslant k < q_i}(O_i[k] = O_j[k])$. The last rule of semantics indicates that the progression of *declass* instruction is similar to an ordinary assignment. By giving the developers with the ability to claim the reference label r in instruction $x := declass(e, r)$, the developers can decide at which reference point the value of the expression becomes declassifiable to a lower security domain. This is different from the external policies defined in [7,18].

4 Security policy and security properties

The security policy formally specifies what can be kept confidential and what can be released by programs. In this section, we present the security policy at the onset. This new policy relies on a user-specific relation

between reference points and legitimately released expressions. Then we define the baseline property, noninterference, and give our relaxed security property for what-dimensional declassification. At last we show how to validate our security property against existing prudent principles of declassification as sanity checks for newly defined properties. Additional conditions required by the new property to comply with certain principles have been found.

4.1 Security policy

In this work, the security policy is a tuple $(\mathcal{D}, \preceq, \sigma, \mathcal{E}, \mathcal{H})$. The set $\mathcal{D}$ of security domains is finite. $\preceq$ is a partial order relation on the security domains. $(\mathcal{D}, \preceq)$ forms a finite lattice of security domains. For example, the most simple two-element security lattice $(\{\text{low}, \text{high}\}, \preceq)$ can be defined with partial order low $\preceq$ high. $\sigma : V \cup \mathcal{I} \cup \mathcal{O} \mapsto \mathcal{D}$ maps each variable and each I/O channel to a security domain. $\sigma(e) \equiv \bigsqcup_{x \in e} \sigma(x)$ denotes the least upper bound of the security domains of variables contained in e. We suppose $\mathcal{I}^{\preceq \ell}$ denotes the set $\{I_i \mid \sigma(I_i) \preceq \ell\}$. The corresponding $p^{\preceq \ell}$ denotes $\{p_i \mid \sigma(I_i) \preceq \ell\}$. Also, $\mathcal{I}^{\succeq \ell}$ denotes $\{I_i \mid \sigma(I_i) \succeq \ell\}$, and $p^{\succeq \ell}$ denotes $\{p_i \mid \sigma(I_i) \succeq \ell\}$. Then, $\mathcal{O}^{\preceq \ell}, \mathcal{O}^{\succeq \ell}, q^{\preceq \ell}, q^{\succeq \ell}$ are similarly defined.

Let R and Exp be the set of reference labels and expressions of a specific program respectively. $\mathcal{E} : R \mapsto 2^{Exp \times \mathcal{D}}$ maps each reference label in R to a set of tuples, which is collected from the *declass* instructions of program. Initially, each reference label r is mapped to $\emptyset$. When we traverse the program and find an instruction $x := declass(e, r)$, if $\sigma(e) \succ \sigma(x)$, a tuple $(e, \sigma(x))$ is added to $\mathcal{E}(r)$.

The security policy related with reference points stipulates the code location where the value of specific expressions can be declassified. When we specify the security property of a program, each run of program is tagged with a specific security domain ℓ, representing an invader's capability of observing data. When a run of program on ℓ reaches a reference label r, we examine the set of currently observable declassified expressions attached to that reference point. Therefore from $\mathcal{E}$ collected in advance, we define the hatch function $\mathcal{H} : R \times \mathcal{D} \mapsto 2^{Exp}$ where $\mathcal{H}(r, \ell) = \{e \mid (e, \ell') \in \mathcal{E}(r) \wedge \ell' \preceq \ell\}$. If $\mathcal{H}(r, \ell) \neq \emptyset$, the expressions in $\mathcal{H}(r, \ell)$ become declassifiable to the security domain ℓ' at the code location of r.

4.2 Baseline property

In order to make the security policy enforceable, we formalize the security properties as concrete security conditions under which the security policy is fulfilled by a specific program. Suppose $I_i \doteq I_j$ represents $\sigma(I_i) = \sigma(I_j) \wedge I_i = I_j$, we first define an indistinguishability relation with respect to a specific security domain.

Definition 1 (ℓ-indistinguishability)**.** For security domain ℓ in $\mathcal{D}$, the *ℓ-indistinguishability* relation $\sim_\ell$ is defined respectively on memory stores, input channels, and output channels of a program:

(1) $\mu \sim_\ell \mu'$, if for all $x \in V$, $\sigma(x) \preceq \ell$ implies $\mu(x) = \mu'(x)$.

(2) $\mathcal{I} \sim_\ell \mathcal{I}'$, if for all i, $\sigma(I_i) \preceq \ell$ implies $I_i \doteq I'_i$.

(3) $\mathcal{O} \sim_\ell \mathcal{O}'$, if for all i, $\sigma(O_i) \preceq \ell$ implies $O_i \doteq O'_i$.

Essentially, both noninterference and the more relaxed security properties are the relations between configurations of any two correlative runs of the same program. Therefore the comparable $\mathcal{I}$ and $\mathcal{I}'$ are the set of channels of the same program, and we require in Definition 1 that $\mathcal{I}$ and $\mathcal{I}'$ in $\sim_\ell$ to have the same domain for each ℓ. Also we know $\sigma(I_i) = \sigma(I'_i)$ holds implicitly for all i. With ℓ-indistinguishability, we can give the definition of noninterference as below.

Definition 2 (Noninterference)**.** Program P satisfies noninterference if for all ℓ in $\mathcal{D}$ and for all $\mathcal{I}, \mathcal{I}', \mu, \mu'$, $\mathcal{O}, \mathcal{O}', \mathcal{O}_f, \mu_f$, if $(\mu, \mathcal{I}, \mathcal{O}, p, q, P) \to^* (\mu_f, \mathcal{I}_f, \mathcal{O}_f, p_f, q_f, \mathbf{skip})$ and $\mu \sim_\ell \mu'$, then there exist $\mathcal{O}'_f, \mu'_f$, such that

(1) $(\mu', \mathcal{I}', \mathcal{O}', p', q', P) \to^* (\mu'_f, \mathcal{I}'_f, \mathcal{O}'_f, p'_f, q'_f, \mathbf{skip})$, and

(2) $\mathcal{I}_f \sim_\ell \mathcal{I}'_f$ can imply $\mu_f \sim_\ell \mu'_f \wedge \mathcal{O}_f \sim_\ell \mathcal{O}'_f$.

From the semantics in Figure 2 we know that no computation can modify the content of input channels. Therefore $\mathcal{I}_f$ is identical to $\mathcal{I}$ and $\mathcal{I}'_f$ is identical to $\mathcal{I}'$. The only difference of initial and final input channels is that the indexes of these channels increase during computation. We know initially each p_i is 0. $\mathcal{I}_f \sim_\ell \mathcal{I}'_f$ indicates that by using p_f and p'_f the actual public inputs of correlative runs are identical.

Also the initial output channels should have each q_i be 0. Each channel of $\mathcal{I}_f$ and $\mathcal{I}'_f$ can be either predefined before execution or related to the intermediate states of run. In this work we only treat each input as indefinite value.

4.3 Relaxed release with reference points

In this section, we define the security property with reference points for the what-dimensional declassification. We first give $(\ell, \mathcal{H})$-indistinguishability to specify the equivalence of declassifiable expressions of ℓ-indistinguishable memory stores.

Definition 3 ($(\ell, \mathcal{H})$-indistinguishability)**.** Suppose memory store μ_r and $\mu'_{r'}$ are respectively within the configuration at reference point r and r'. μ_r and $\mu'_{r'}$ are $(\ell, \mathcal{H})$-*indistinguishable*, denoted by $\mu_r \sim_\ell^{\mathcal{H}} \mu'_{r'}$, if $\mu_r \sim_\ell \mu'_{r'}$ and $\mathcal{H}(r, \ell) = \mathcal{H}(r', \ell)$ and $\mu_r(e) = \mu'_{r'}(e)$ for all e in $\mathcal{H}(r, \ell)$.

Different from the ℓ-indistinguishability, $(\ell, \mathcal{H})$-indistinguishability relation is restricted on memory stores because the value of expression is not directly related with elements of I/O channels except through I/O commands. In addition, this relation requires the declassifiable expressions at r and r' to have identical form and values. Later on, by constraining the precondition of property with $(\ell, \mathcal{H})$-indistinguishability we can rule out the irrelevant pair of traces that cause the attacker to learn the variation of value of declassifiable expression.

The declassification property requires that when a reference point labeled with r is reached and $\mathcal{H}(r, \ell) \neq \emptyset$, the expressions in $\mathcal{H}(r, \ell)$ become declassifiable simultaneously. Therefore these expressions can be legally released from the specified time onwards. To simplify the representation of relaxed property, we present another prerequisite concept, i.e. hatchless sequence of execution.

Definition 4 (Hatchless Sequence)**.** The hatchless sequence is a member of relation $\twoheadrightarrow$ on configurations, such that $(\mu_s, \mathcal{I}_s, \mathcal{O}_s, p_s, q_s, r_s : P_s) \twoheadrightarrow (\mu_t, \mathcal{I}_t, \mathcal{O}_t, p_t, q_t, r_t : P_t)$ if

(1) $(\mu_s, \mathcal{I}_s, \mathcal{O}_s, p_s, q_s, r_s : P_s) \rightarrow (\mu_1, \mathcal{I}_1, \mathcal{O}_1, p_1, q_1, r_1 : P_1) \rightarrow \cdots \rightarrow (\mu_k, \mathcal{I}_k, \mathcal{O}_k, p_k, q_k, r_k : P_k) \rightarrow (\mu_t, \mathcal{I}_t, \mathcal{O}_t, p_t, q_t, r_t : P_t)$, and

(2) $\mathcal{H}(r_j, \ell)$ is empty for all $1 \leqslant j \leqslant k$, and

(3) either $\mathcal{H}(r_t, \ell)$ is not empty or P_t is the instruction **skip**.

If the inputs of the two correlative runs are ℓ-indistinguishable, each time a run reaches a reference label r where the expressions in $\mathcal{H}(r, \ell)$ become declassifiable, there should be some $r' \in R$ passed by the correlative run, and satisfying that if the precondition of property implies $\mu_r \sim_\ell^{\mathcal{H}} \mu'_{r'}$, the final states of the runs are ℓ-indistinguishable. When an attacker detects the violation of final ℓ-indistinguishability and derive some confidentiality, the contrapositive indicates that some $\mu_r \sim_\ell^{\mathcal{H}} \mu'_{r'}$ is violated. Otherwise the program is insecure. For a secure program, the property should be able to restrain the released confidential data to the information of declassifiable expressions, and forbid the release of other confidential data. In order to sustain that the violation of the postcondition of property is caused by the variation of declassifiable expressions, i.e. $\mathcal{H}(r, \ell) \neq \mathcal{H}(r', \ell)$ or $\mu_r(e) \neq \mu'_{r'}(e)$ for some e in $\mathcal{H}(r, \ell)$, we require to derive $\mu_r \sim_\ell \mu'_{r'}$. The relaxed property is formalized as follows.

Definition 5 (Relaxed Release with Reference Points, R3P)**.** Program P is secure with respect to R3P if for all ℓ in $\mathcal{D}$ and for all $\mathcal{I}, \mathcal{I}', \mu, \mu', \mathcal{O}, \mathcal{O}', \mathcal{O}_k, \mu_k$, if $(\mu, \mathcal{I}, \mathcal{O}, p, q, P) \twoheadrightarrow \cdots \twoheadrightarrow (\mu_k, \mathcal{I}_k, \mathcal{O}_k, p_k, q_k, \mathbf{skip})$ and $\mu \sim_\ell \mu'$, then there exists $\mathcal{O}'_k, \mu'_k$, such that

(1) $(\mu', \mathcal{I}', \mathcal{O}', p', q', P) \twoheadrightarrow \cdots \twoheadrightarrow (\mu'_k, \mathcal{I}'_k, \mathcal{O}'_k, p'_k, q'_k, \mathbf{skip})$, and

(2) for all $j (1 \leqslant j \leqslant k)$, if $\mathcal{I}_j \sim_\ell \mathcal{I}'_j$, then $\bigwedge_{1 \leqslant i < j} \mu_i \sim_\ell^{\mathcal{H}} \mu'_i$ implies $\mu_j \sim_\ell \mu'_j \wedge \mathcal{O}_j \sim_\ell \mathcal{O}'_j$.

Before the first reference point with non-empty $\mathcal{H}(r, \ell)$ is met, that is when $j = 1$, the executed subprogram should comply with noninterference. The consequent of one step, i.e. $(\mu_j \sim_\ell \mu'_j)$, can fulfil part of precondition of the next step $(\mu_j \sim_\ell^{\mathcal{H}} \mu'_j)$. Therefore when the two sets of newly declassifiable expressions are identical and each pair of these expressions are equal within the correlative memory stores in each step, the program satisfying R3P should have $\bigwedge_{1 \leqslant j \leqslant k} (\mu_j \sim_\ell \mu'_j \wedge \mathcal{O}_j \sim_\ell \mathcal{O}'_j)$, otherwise R3P is violated. This definition coincides with the principle that the security property can only permit the release of declassifiable expressions and the violation of this property can only be caused by the release

of information other than declassifiable expressions. The information release is then characterized to be a condition where an expression e is allowed to be released by an instruction of program only when some reference point labeled with r locates before that instruction and $e \in \mathcal{H}(r, \ell)$.

Remark 1 (Loop related decision)**.** The security property is more complicated when we meet loops within finite runs. Suppose in the following examples we use the binary security lattice low$\preceq$high:

$$r : \text{while}(h)\{l := \textit{declass}(h + l, r); h := h - 1; \}; \quad (1)$$

$$\text{while}(h)\{r : l := \textit{declass}(h + l, r); h := h - 1; \}; \quad (2)$$

$$l := h; \text{while}(l)\{r : l := \textit{declass}(h, r); h := h - 1; \}. \quad (3)$$

In example (1) we have $\mathcal{H}(r, \text{low}) = \{h + l\}$, therefore the initial value of $h + l$ is declassifiable. Moreover, the initial $\mu \sim_{\text{low}} \mu'$ ensures $l = l'$. Consequently, we can derive the equality on initial values of h of the correlative runs and the program is decided to be secure with respect to our property. Example (2) is not secure for reason that, with a different initial value of h, the second run of program may not reach the end after the k-th period. Moreover, when both runs of program reach r for the second time, $\mu_2 \sim_{\text{low}} \mu'_2$ is violated since the initial value of h is added to l. The example (3) is also insecure considering that $\mu_1 \nsim_{\text{low}} \mu'_1$, that is $l \neq l'$ at the first occurrence of r where $\mathcal{H}(r, \text{low})$ is not empty. The above cases assume finite loops. On the other hand, the infinite loops are out of the scope of our decision therefore R3P is a termination-insensitive property [25].

Remark 2 (Difference between R3P and WERP)**.** WERP [7, Def. 6] is the most relevant security property to R3P. The reference points labeled with non-empty hatches separate a run of program into several periods. R3P decides the condition $\mu_j \sim_\ell \mu'_j$ and $\mathcal{O}_j \sim_\ell \mathcal{O}'_j$ at the end of each period, while the bisimulation-based WERP decides the equivalence by each computation step. The difference between the two properties can be illustrated by the examples in Table 1. For C_1 we have $\mathcal{H}(r_1, \text{low}) = \{h_1 + h_2\}$ and $\mathcal{H}(r_2, \text{low}) = \emptyset$. C_1 is insecure because the security condition of R3P requires that if the respective initial values of $h_1 + h_2$ of the two correlative runs are equal, i.e. $h_1 + h_2 = h'_1 + h'_2$, there should be $l = l'$ at the final state. However, this is violated by C_1. On the other hand, with the hatch invalidation function ih of WERP, we know $ih(C_1, \{(h_1 + h_2, \text{low})\}) = \emptyset$. Therefore the property requires $l = l'$ at reference label r_2 and the final state, while it is only violated at the final state. C_2 is used to explain that the reference points have the right locations to make expressions declassifiable. C_2 violates R3P at r even if $\mathcal{H}(r, \text{low}) = \{h\}$. The variants of C_2 whose instruction $l := h$ is in the scope of declassifiable h, e.g. C_3 and C_5, are secure no matter whether this instruction is put before or after the intentional *declass* primitive. The periodical decision of R3P is less restrictive than the stepwise decision of WERP. For instance, C_4 is decided to be secure by R3P, because the inspection of intermediate $\mu_j \sim_\ell \mu'_j$ is taken at the end of each period. The intermediate hatchless states are not inspected, and thus the final states do not leak h_2. Contrarily, WERP requires the intermediate states after $l := h_2$ to satisfy $l = l' \wedge h = h'$, but this is not fulfilled. C_6 is insecure according to R3P because the final l of C_6 leaks h_2 instead of the declassifiable h. Similarly, $h = h'$ of intermediate states cannot imply final $l = l'$ by WERP. C_7 is used to explain whether the enforcements of respective properties can capture the value-dependent security condition, as explained in Section 6. C_7 is secure according to R3P because even though the initial $h_1 - h_1 = h'_1 - h'_1$ cannot exclude any irrelevant trace, the final $l = l'$ is always satisfied. On the other hand, WERP is also satisfied because $ih(h_1 := h_2, \{(h_1 - h_1, \text{low})\}) = \emptyset$ and we have $l = l'$ at r_2 and $\{l := h_1 - h_1\}R^{\emptyset}\{l' := h'_1 - h'_1\}$.

Remark 3 (Reduction from R3P to relaxed noninterference and where-security)**.** R3P can be reduced to the relaxed noninterference [8] if we require the reference points labeled with non-empty hatch can only be the initial one. Conclusively, what we deliberately release is the initial value of the declassifiable expressions. It means μ and μ' are respectively reduced to μ_1 and μ'_1, and $k = 2$. In Section 6 we will evaluate our enforcement for relaxed noninterference reduced from R3P compared with the previous approach based on another safety verification [8]. The where-security [9] is a security property describing where in the code the intentional released data become declassifiable. The location of release is actually at each *declass* command. That means if we set the reference label r in each *declass*(e, r) to be the

Table 1 Property comparison with WERP

Case	Commands	WERP	R3P
C_1	$r_1 : h_1 := 0; r_2 : l := declass(h_1 + h_2, r_1);$	×	×
C_2	$l := h; r : l := declass(h, r);$	×	×
C_3	$r : l := h; l := declass(h, r);$	✓	✓
C_4	$r : l := h_2; l := declass(h, r);$	×	✓
C_5	$r : l := declass(h, r); l := h;$	✓	✓
C_6	$r : l := declass(h, r); l := h_2;$	×	×
C_7	$r_1 : h_1 := h_2; r_2 : l := declass(h_1 - h_1, r_1)$	✓	✓

reference label of the command itself, R3P can be reduced to where-security. In this case, when we meet a command $r:\ x := declass(e, r')$, we always have $r = r'$. Meanwhile, $\mathcal{H}(r, \ell)$ will be empty if r is not a reference label of a *declass* command. In summary, compared with these properties enforceable through automated verification, our new security property is exceedingly generalized.

4.4 Prudent principles for declassifications

To evaluate the success of an information release policy, a set of standard principles has to be implemented that is in compliance with every security property. Sabelfeld and Sands [5] proposed four basic prudent principles for declassification policies as sanity checks for the existing and newly defined security properties-*semantic consistency*, *conservativity*, *monotonicity of release*, and *non-occlusion*. Several other principles were proposed by Mantel et al, such as *compositionality* [18], *relaxation*, *noninterference up-to*, *(weakly) persistence*, and *protection* [10]. Although these principles are not mandatory to be complied with by the security properties, they serve as important accordance for the assessment of attacker model and choice of security policy. In this section, we focus on the compliance of R3P with both the basic principles and the principles given in Ref. [10]. Let $P[C]$ be a program with C as a subprogram, and $P[C'/C]$ denotes a program derived by substituting each occurrence of C in P with C'.

Theorem 1 (Semantic consistency)**.** Suppose command C and C' are declassification-free and semantically equivalent on same domain of configuration. If program $P[C]$ satisfies R3P, then $P[C'/C]$ also satisfies R3P.

Proof. First, we define a relation $\mathbb{R}^\ell$ on programs. The relation is parameterized by security domain ℓ. For any program P and P', $(P, P') \in \mathbb{R}^\ell$ if and only if,

$$\left[\begin{array}{c} \forall \mathcal{I}, \mathcal{I}', \mu, \mu', \mathcal{O}, \mathcal{O}', \mathcal{O}_k, \mu_k : (\mu, \mathcal{I}, \mathcal{O}, p, q, P) \twoheadrightarrow \cdots \twoheadrightarrow (\mu_k, \mathcal{I}_k, \mathcal{O}_k, p_k, q_k, \mathbf{skip}) \wedge \mu \sim_\ell \mu' \\ \Rightarrow \left[\begin{array}{l} \exists \mathcal{O}'_k, \mu'_k : (\mu', \mathcal{I}', \mathcal{O}', p', q', P') \twoheadrightarrow \cdots \twoheadrightarrow (\mu'_k, \mathcal{I}'_k, \mathcal{O}'_k, p'_k, q'_k, \mathbf{skip}) \\ \wedge \bigwedge_{1 \leqslant j \leqslant k} (\mathcal{I}_j \sim_\ell \mathcal{I}'_j \Rightarrow (\bigwedge_{1 \leqslant i < j} \mu_i \sim_\ell^{\mathcal{H}} \mu'_i \Rightarrow \mu_j \sim_\ell \mu'_j \wedge \mathcal{O}_j \sim_\ell \mathcal{O}'_j)) \end{array}\right] \end{array}\right].$$

It is obvious that P satisfies R3P if $(P, P) \in \mathbb{R}^\ell$ for all ℓ in $\mathcal{D}$. We then show $(P, P[C'/C]) \in \mathbb{R}^\ell$ for all ℓ in $\mathcal{D}$. Suppose C is executed in a hatchless sequence $(\mu_x, \mathcal{I}_x, \mathcal{O}_x, p_x, q_x, P_x) \twoheadrightarrow (\mu_{x+1}, \mathcal{I}_{x+1}, \mathcal{O}_{x+1}, p_{x+1}, q_{x+1}, P_{x+1})$, we have $(\mu_x, \mathcal{I}_x, \mathcal{O}_x, p_x, q_x, P_x[C'/C]) \twoheadrightarrow (\mu_{x+1}, \mathcal{I}_{x+1}, \mathcal{O}_{x+1}, p_{x+1}, q_{x+1}, P_{x+1})$ in that C and C' are semantically equivalent. Let -/+ be respectively for before/after the substitution. Because C and C' are declassification-free and the substitution of C' for C does not change the reference label of C, if P satisfies R3P, we can ensure $\bigwedge_{1 \leqslant j \leqslant k} \mu_j \ \sim_\ell^{\mathcal{H}} \ \mu_j'^{+}$. Therefore for all ℓ in $\mathcal{D}$, $(P, P) \in \mathbb{R}^\ell$ implies $(P, P[C'/C]) \in \mathbb{R}^\ell$. The transitivity of $\mathbb{R}^\ell$ can be proved by instantiating the precondition of $(P_2, P_3) \in \mathbb{R}^\ell$ based on $(P_1, P_2) \in \mathbb{R}^\ell$. From the symmetry of the substitution we have $(P[C'/C], P) \in \mathbb{R}^\ell$ for all ℓ in $\mathcal{D}$, and from the transitivity we have $(P[C'/C], P[C'/C]) \in \mathbb{R}^\ell$ for all ℓ in $\mathcal{D}$, i.e., $P[C'/C]$ satisfies R3P.

The next two principles show the monotonicity of the security property. Intuitively speaking, more declassifications in a program result in more relaxed security condition. Newly added declassification will not make a secure program insecure. In an extreme situation, the absence of declassification in programs should make the security property to ensure the base-line noninterference.

Theorem 2 (Conservativity)**.** If program P satisfies R3P and P is declassification-free, then P satisfies noninterference.

Proof. Because P is declassification-free, $\mathcal{E}(r)$ is empty for all r in R. Then we know $\mathcal{H}(r,\ell)$ is empty for all ℓ in $\mathcal{D}$ and r in R. Therefore any execution of P is a hatchless sequence, and R3P is reduced to have $k=1$. That is to say, there is no such i to form relation $\bigwedge_{1\leqslant i<j}\mu_i \sim_\ell^{\mathcal{H}} \mu'_i$. This precondition is reduced to *true* and the most inner part of the postcondition becomes $\mathcal{I}_1 \sim_\ell \mathcal{I}'_1 \Rightarrow \mu_1 \sim_\ell \mu'_1 \wedge \mathcal{O}_1 \sim_\ell \mathcal{O}'_1$. Hence, P satisfies noninterference.

The monotonicity of release is restricted for R3P to add declassifications to reference points labeled with r where $\mathcal{H}(r,\ell)\neq\emptyset$ for all $\ell \succeq \sigma(x)$. Intuitively speaking, we should avoid the newly added declassifiable expression to introduce additional most inner postcondition $\mu_j \sim_\ell \mu'_j \wedge \mathcal{O}_j \sim_\ell \mathcal{O}'_j$.

Theorem 3 (Monotonicity of release)**.** If program $P[x:=e]$ satisfies R3P and $\mathcal{H}(r,\ell)$ is nonempty for all $\ell \succeq \sigma(x)$, the program $P[x := \mathit{declass}(e,r)/x:=e]$ also satisfies R3P.

Proof. First, we simplify the security condition of R3P. From the definition of $(\ell,\mathcal{H})$-indistinguishability we have $(P,P')\in\mathbb{R}^\ell$ if and only if

$$\left[\begin{array}{l} \forall \mathcal{I},\mathcal{I}',\mu,\mu',\mathcal{O},\mathcal{O}',\mathcal{O}_k,\mu_k : (\mu,\mathcal{I},\mathcal{O},p,q,P) \twoheadrightarrow \cdots \twoheadrightarrow (\mu_k,\mathcal{I}_k,\mathcal{O}_k,p_k,q_k,\mathbf{skip}) \wedge \mu\sim_\ell \mu' \Rightarrow \\ \left[\begin{array}{l} \exists \mathcal{O}'_k,\mu'_k : (\mu',\mathcal{I}',\mathcal{O}',p',q',P') \twoheadrightarrow \cdots \twoheadrightarrow (\mu'_k,\mathcal{I}'_k,\mathcal{O}'_k,p'_k,q'_k,\mathbf{skip}) \wedge \bigwedge_{1\leqslant j\leqslant k}(\mathcal{I}_j \sim_\ell \mathcal{I}'_j \Rightarrow \\ \quad (\bigwedge_{1\leqslant i<j}(\mathcal{H}(r_i,\ell)=\mathcal{H}(r'_i,\ell)\wedge \bigwedge_{e\in\mathcal{H}(r_i,\ell)}(\mu_i(e)=\mu'_i(e))) \Rightarrow \mu_j\sim_\ell \mu'_j \wedge \mathcal{O}_j \sim_\ell \mathcal{O}'_j)) \end{array}\right] \end{array}\right].$$

Suppose $x := e$ is executed in the hatchless sequence $(\mu_x,\mathcal{I}_x,\mathcal{O}_x,p_x,q_x,P_x) \twoheadrightarrow (\mu_{x+1},\mathcal{I}_{x+1},\mathcal{O}_{x+1},p_{x+1},q_{x+1},P_{x+1})$. From the semantics in Figure 2, we know $(\mu_x,\mathcal{I}_x,\mathcal{O}_x,p_x,q_x,P_x[x := \mathit{declass}(e,r)/x := e]) \twoheadrightarrow (\mu_{x+1},\mathcal{I}_{x+1},\mathcal{O}_{x+1},p_{x+1},q_{x+1},P_{x+1})$ and if $\sigma(x)\prec\sigma(e)$ then $(e,\sigma(x))\in\mathcal{E}(r)$ after the substitution.

1. If $\sigma(e)\preceq\sigma(x)$, then from the definition of $\mathcal{E}$ we know $\mathcal{E}^+(r)=\mathcal{E}^-(r)$ for any specific r. Therefore $\mathcal{H}^+(r,\ell)=\mathcal{H}^-(r,\ell)$ for all ℓ in $\mathcal{D}$, and we have $\bigwedge_{1\leqslant i<k}\mu_i\sim_\ell^{\mathcal{H}^+}\mu'_i$ is equivalent to $\bigwedge_{1\leqslant i<k}\mu_i\sim_\ell^{\mathcal{H}^-}\mu'_i$.

2. If $\sigma(x)\prec\sigma(e)$, there are two cases:

(a) If $\ell\prec\sigma(x)$, then even though $(e,\sigma(x))\in\mathcal{E}^+(r)$, we still have $\mathcal{H}^+(r,\ell)=\mathcal{H}^-(r,\ell)$ and $\bigwedge_{1\leqslant i<k}\mu_i\sim_\ell^{\mathcal{H}^+}\mu'_i$ equivalent to $\bigwedge_{1\leqslant i<k}\mu_i\sim_\ell^{\mathcal{H}^-}\mu'_i$.

(b) If $\sigma(x)\preceq\ell$, we have $\mathcal{H}(r,\ell)\neq\emptyset$. Then e is added to $\mathcal{H}(r,\ell)$ by substitution and $\mathcal{H}^+(r,\ell)=\mathcal{H}^-(r,\ell)\cup\{e\}$. If $\mathcal{H}^+(r,\ell)=\mathcal{H}^+(r',\ell)\wedge\bigwedge_{e_0\in\mathcal{H}^+(r,\ell)}(\mu_j(e_0)=\mu'_j(e_0))$ then $\mathcal{H}^-(r,\ell)=\mathcal{H}^-(r',\ell)\wedge\bigwedge_{e_0\in\mathcal{H}^-(r,\ell)}(\mu_j(e_0)=\mu'_j(e_0))$. Because $\mathcal{H}^-(r,\ell)$ is not empty, we know the substitution would not add new most inner postcondition $\mu_j\sim_\ell\mu'_j$.

Similarly the substitution cannot cause new output therefore $\mathcal{O}_j\sim_\ell\mathcal{O}'_j$ is derived after substitution. In summary, $(P[x := \mathit{declass}(e,r)/x:=e], P[x := \mathit{declass}(e,r)/x:=e])\in\mathbb{R}^\ell$ for all ℓ in $\mathcal{D}$.

The restriction to require $\mathcal{H}^-(r,\ell)$ to be nonempty can be avoided when each expression is exclusively permitted to become declassifiable at the initial state, i.e., when R3P is reduced to relaxed noninterference (see Remark 3, Subsection 4.3).

The fourth principle, non-occlusion, justifies the fact that the presence of a declassification operation cannot make an illegal leakage undetectable. R3P does not comply with this principle. For instance, consider the following example:

$$r_1 : \text{if } \mathit{true} \text{ then } l := h*h \text{else} r_2 : l := \mathit{declass}(h*h, r_x);$$

If the specific r_x is r_1, the de facto leakage on the *then*-branch is covered up by the precondition $\mu\sim_{\text{low}}^{\{h*h\}}\mu'$ where $h*h$ is collected syntactically from the unreachable *else*-branch. On the contrary, if r_x is r_2, the covert flow is not masked and still detectable.

The prudent principles proposed by Lux and Mantel [10] for the what-dimensional declassification were proved to be suitable for the stepwise bisimulation-based security condition WHERE&WHO, but have not been adequately discussed on other declassification policies, except in our previous work [9]. *Relaxation* cannot be fulfilled by R3P because the postconditions of R3P contain more ℓ-indistinguishability relation between intermediate states. Consider the following example

$$l := h; r : l := \mathit{declass}(h,r); l := 0.$$

The program satisfies noninterference but violates R3P because initially $h \neq h'$ and then $\mu_r \nsim_{\text{low}} \mu'_r$. Lux and Mantel have also pointed out that the relaxation principle can subsume conservativity, however, here we observe that this subsumption depends on the bisimulation specification of strong security. In this work the relaxation can be fulfilled if we refine the noninterference in Definition 2 to include periodical decision of $\mathcal{I}_j \sim_\ell \mathcal{I}'_j \Rightarrow \mu_j \sim_\ell \mu'_j \wedge \mathcal{O}_j \sim_\ell \mathcal{O}'_j$. We then show that R3P is *noninterference up-to*, which is another principle that can subsume conservativity.

Theorem 4 (Noninterference up-to)**.** If program P satisfies R3P, we have $\mu \sim_\ell \mu' \Rightarrow (\mathcal{I}_1 \sim_\ell \mathcal{I}_1 \Rightarrow \mu_1 \sim_\ell \mu'_1 \wedge \mathcal{O}_1 \sim_\ell \mathcal{O}'_1)$ for all ℓ in $\mathcal{D}$.

Proof. From the satisfaction of R3P and $\mu \sim_\ell \mu'$, we have $\bigwedge_{1\leqslant j\leqslant k}(\mathcal{I}_j \sim_\ell \mathcal{I}'_j \Rightarrow (\bigwedge_{1\leqslant i<j} \mu_i \sim_\ell^{\mathcal{H}} \mu'_i \Rightarrow \mu_j \sim_\ell \mu'_j \wedge \mathcal{O}_j \sim_\ell \mathcal{O}'_j))$. Then similar to the proof of conservativity, when $j = 1$ the proposition is proved. The subsumption is obvious because conservativity is derived from noninterference up-to by restricting the program to be declassification-free.

The last principle for what-dimensional declassification is *persistence*. It means all the reachable subprograms of a secure program should be secure as well. R3P cannot comply with this principle because of the periodical nature of decision. For instance, the program C_5 in Table 1 satisfies R3P, but the reachable subprogram $l := h$ violates R3P. There are actually additional conditions to hold the reachable subprograms secure. To reveal the conditions, we propose a new principle, *conditional persistence*. Similar to the monotonicity of release, the new principle can only be fulfilled when the reachable subprograms are restricted to the ones starting from reference label r where $\mathcal{H}(r, \ell)$ is not empty for a specific ℓ. The new principle is also different from *weakly persistence* [10] which restricts the start points at the *declass* commands instead of reference labels.

Theorem 5 (Conditional persistence)**.** Suppose that program P satisfies R3P, and for all $\ell, \mathcal{I}, \mu$ there exists a sequence $(\mu, \mathcal{I}, \mathcal{O}, p, q, P) \twoheadrightarrow^* (\mu_x, \mathcal{I}_x, \mathcal{O}_x, p_x, q_x, P_x)$. If $\mu_x \sim_\ell \mu'_x$ can imply $\mu \sim_\ell \mu' \wedge \bigwedge_{1\leqslant i\leqslant x} \mu_i \sim_\ell^{\mathcal{H}} \mu'_i$, then P_x satisfies R3P as well.

Proof. If P is P_x, then P_x satisfies R3P trivially. If P has a form $P_y; P_x$ and P_x is like $r_x : C_x; P_z$, then from $\mu_x \sim_\ell \mu'_x$ we have $\mu \sim_\ell \mu'$ for P. Then because P satisfies R3P, we have $\bigwedge_{1\leqslant j\leqslant k}(\mathcal{I}_j \sim_\ell \mathcal{I}'_j \Rightarrow (\bigwedge_{1\leqslant i<j} \mu_i \sim_\ell^{\mathcal{H}} \mu'_i \Rightarrow \mu_j \sim_\ell \mu'_j \wedge \mathcal{O}_j \sim_\ell \mathcal{O}'_j))$. For each $j (x < j \leqslant k)$, if $\mathcal{I}_j \sim_\ell \mathcal{I}'_j$ then we have $\bigwedge_{1\leqslant i<j} \mu_i \sim_\ell^{\mathcal{H}} \mu'_i \Rightarrow \mu_j \sim_\ell \mu'_j \wedge \mathcal{O}_j \sim_\ell \mathcal{O}'_j$. If $\bigwedge_{x<i<j} \mu_i \sim_\ell^{\mathcal{H}} \mu'_i$, then from $\mu_x \sim_\ell \mu'_x \Rightarrow \bigwedge_{1\leqslant i\leqslant x} \mu_i \sim_\ell^{\mathcal{H}} \mu'_i$ we have $\bigwedge_{1\leqslant i<j} \mu_i \sim_\ell^{\mathcal{H}} \mu'_i$. And then we have $\mu_j \sim_\ell \mu'_j \wedge \mathcal{O}_j \sim_\ell \mathcal{O}'_j$. Therefore $\bigwedge_{x<j\leqslant k}(\mathcal{I}_j \sim_\ell \mathcal{I}'_j \Rightarrow (\bigwedge_{x<i<j} \mu_i \sim_\ell^{\mathcal{H}} \mu'_i \Rightarrow \mu_j \sim_\ell \mu'_j \wedge \mathcal{O}_j \sim_\ell \mathcal{O}'_j))$, and that is to say P_x satisfies R3P.

In summary, we have discussed in this section the compliance of R3P to the existing prudent principles for declassification. We have also figured out the restriction on our policy to comply with the prudent principles by the proof of *monotonicity of release* and the development of *conditional persistence*. The results are summarized in Table 2.

5 Enforcement with reachability analysis

In this section, we adopt the enforcement for where-dimensional security property [9] incidentally for what-dimensional R3P. We give a model transformation of symbolic pushdown system based on self-composition. This transformation can reduce the check on satisfaction of R3P to a detection on whether an illegal-flow state is unreachable in the model after transformation.

5.1 Model construction

We leverage symbolic pushdown system [26] as the abstract model of program. The unbounded stack of pushdown system makes it a natural model for sequential procedural programs. In this work, the procedures are mainly used to model the output operations. This treatment can delay the compensation of store-match operations after the basic self-composition. Symbolic pushdown system is a compact representation of pushdown system encoding the abstract variables and computations symbolically. We refer its definition to [27, Def. 3]. The nodes of control flow graph are denoted by the explicit stack

Table 2 Compliance to prudent principles

Principle	Compliance
Semantic consistency	✓
Conservativity	✓
Monotonicity of release	✓* (restricted)
Non-occlusion	×
Relaxation	×
Noninterference up-to	✓
Persistence	×
Conditional persistence	✓

Algorithm 1. Reachability with $post^*$

```
1: procedure Reachable(P, γ0)                                  //P: SPDS; γ0: a CFG node
2:   Generate the P-automaton A = ({s,t}, Γ, {(s, r_init, t)}, {s}, {t})
3:   trans ← {(s, r_init, t)}
4:   while s ~γ~> t found in trans and trans not saturated do
5:     for all ⟨γ⟩ ↪ ⟨ε⟩ ∈ Δ do
6:       trans ← trans ∪ {(s, ε, t)}
7:     for all ⟨γ⟩ ↪ ⟨γ'⟩ ∈ Δ do
8:       trans ← trans ∪ {(s, γ', t)}
9:     for all ⟨γ⟩ ↪ ⟨γ'γ''⟩ ∈ Δ do
10:      trans ← trans ∪ {(s, γ', t_γ'), (t_γ', γ'', t)}
11:  end while
12:  A_post* := ({s,t} ∪ {t_γ' | ⟨γ⟩ ↪ ⟨γ'γ''⟩ ∈ Δ}, Γ, trans, {s}, {t})
13:  return whether A_post* can accept a configuration whose top stack symbol is γ0
14: end procedure
```

symbols in Γ. The relation $\mathcal{R}$ specifies the environment transformer that directs a single step of symbolic computation according to the pushdown rule. The back-end algorithm we used for reachability analysis is the saturation algorithm $post^*$ adapted from [28, Sec. 2], sketched in Algorithm 1. ϵ stands for the empty stack symbol when $n = 0$. The reachability analysis of a node of control-flow graph (CFG), e.g. γ_0, actually checks whether the derived $\mathcal{P}$-automaton $\mathcal{A}_{post^*}$ can accept a configuration whose top stack symbol is γ_0. $\overset{\gamma}{\rightsquigarrow}$ denotes the relation $(\overset{\epsilon}{\rightarrow})^* \overset{\gamma}{\rightarrow} (\overset{\epsilon}{\rightarrow})^*$. r_{init} denotes the entry node of CFG. The symbolic algorithm is given in detail in [26, Sec. 3.3.3].

We propose the procedure $\mathfrak{A}$ in Algorithm 2 to abstract the program in our presentation language with symbolic pushdown system. The procedure also adopts the abstraction of simple imperative language without I/O [29]. The abstraction corresponds to a specific security domain ℓ. From the syntax of language in Figure 1, we know each instruction has a reference label in R. The reference labels have correspondence with the nodes of CFG. The primary procedure GenTrans recursively derives the set of symbolic pushdown rules for the pushdown system. In this procedure, rt derives a logical relation to express the retainment on both the value of global variables and the value of local variables of the procedure locating the pushdown rule. For a memory store μ, $rt(\mu)$ means $\bigwedge_{x\in V}(\mu'(x) = \mu(x))$, where μ and μ' stand for the memory store attached to the configuration before and after the computation directed by the pushdown rule. Let $rt(a, b)$ be $rt(a) \wedge rt(b)$ and $rt(\mu \setminus \{x_0\})$ be $\bigwedge_{x\in V\setminus\{x_0\}}(\mu'(x) = \mu(x))$.

When modeling the intermediate I/Os, (1) the difference between confidential and public channels, and (2) the benefit obtained by the store-match pattern from the modeling of I/Os, are to be considered. The confidential channels in $\mathcal{I}^{\succ\ell}$ are omitted and the confidential inputs as an assignment from an indefinite value $\perp$ to a variable are modeled. The confidential output channels in $\mathcal{O}^{\succ\ell}$ are also omitted and the confidential outputs can be modeled similar to a rule of **skip**. The public channels need to be modeled explicitly. The abstraction of public inputs is trivially derived from the operational semantics as the data on the public input channels in $\mathcal{I}^{\preceq\ell}$ are knowable to the attacker. The abstraction of public outputs is exceedingly complicated. For each public output channel $O_i \in \mathcal{O}^{\preceq\ell}$, it consists of two pushdown rules:

Algorithm 2. Model Abstraction $\mathfrak{A}$

1: **procedure** $\mathfrak{A}(C, \ell)$ //C: sequence of commands; ℓ: a security domain
2: Generate an exit label γ_{exit}, $(\gamma_{exit} \notin R)$
3: **return** $(dom(\mathcal{I}^{\preceq\ell} \times p^{\preceq\ell} \times \mathcal{O}^{\preceq\ell} \times q^{\preceq\ell}), R \times dom(\mu), \text{GenTrans}(C, \gamma_{exit}, \ell))$
4: **end procedure**
5: **procedure** $\text{GenTrans}(C,\gamma_{exit},\ell)$ //γ_{exit}: exit node for command C
6: **if** C is $r_1 : c_1; r_2 : c_2; \ldots; r_k : c_k$ **then** //Sequence of commands
7: **return** $\text{GenTrans}(r_1 : c_1,\ r_2,\ \ell) \cup \text{GenTrans}(r_2 : c_2; \ldots; r_k : c_k;,\ \gamma_{exit},\ \ell)$
8: **else if** C is r_i : **skip then return** $\{\langle r_i\rangle \hookrightarrow \langle \gamma_{exit}\rangle\ rt(\mu, \mathcal{I}^{\preceq\ell}, p^{\preceq\ell}, \mathcal{O}^{\preceq\ell}, q^{\preceq\ell})\}$ // **skip**
9: **else if** C is $r_i : x := e$ **then return** $\{\langle r_i\rangle \hookrightarrow \langle \gamma_{exit}\rangle\ (x' = e) \wedge rt(\mu \setminus \{x\}, \mathcal{I}^{\preceq\ell}, p^{\preceq\ell}, \mathcal{O}^{\preceq\ell}, q^{\preceq\ell})\}$ //Assignment
10: **else if** C is r_i : (**if** e **then** $r_\mathsf{t} : c_\mathsf{t}; C'_\mathsf{t}$ **else** $r_\mathsf{f} : c_\mathsf{f}; C'_\mathsf{f}$) **then** //**if-else**
11: **return** $\{\langle r_i\rangle \hookrightarrow \langle r_\mathsf{t}\rangle\ rt(\mu, \mathcal{I}^{\preceq\ell}, p^{\preceq\ell}, \mathcal{O}^{\preceq\ell}, q^{\preceq\ell}) \wedge e, \langle r_i\rangle \hookrightarrow \langle r_\mathsf{f}\rangle\ rt(\mu, \mathcal{I}^{\preceq\ell}, p^{\preceq\ell}, \mathcal{O}^{\preceq\ell}, q^{\preceq\ell}) \wedge \neg e\}$
$\cup \text{GenTrans}(r_\mathsf{t} : c_\mathsf{t}; C'_\mathsf{t}, \gamma_{exit}, \ell) \cup \text{GenTrans}(r_\mathsf{f} : c_\mathsf{f}; C'_\mathsf{f}, \gamma_{exit}, \ell)$
12: **else if** C is r_i : (**while** e **do** $r_j : c_j; C'$) **then** //**while**
13: **return** $\{\langle r_i\rangle \hookrightarrow \langle r_j\rangle\ rt(\mu, \mathcal{I}^{\preceq\ell}, p^{\preceq\ell}, \mathcal{O}^{\preceq\ell}, q^{\preceq\ell}) \wedge e, \langle r_i\rangle \hookrightarrow \langle \gamma_{exit}\rangle\ rt(\mu, \mathcal{I}^{\preceq\ell}, p^{\preceq\ell}, \mathcal{O}^{\preceq\ell}, q^{\preceq\ell}) \wedge \neg e\}$
$\cup \text{GenTrans}(r_j : c_j; C', r_i, \ell)$
14: **else if** C is $r_i : input(x, \mathcal{I}_i)$ **then** //Confidential and public inputs
15: **if** $\sigma(\mathcal{I}_i) \succ \ell$ **then return** $\{\langle r_i\rangle \hookrightarrow \langle \gamma_{exit}\rangle\ (x' = \bot) \wedge rt(\mu \setminus \{x\}, \mathcal{I}^{\preceq\ell}, p^{\preceq\ell}, \mathcal{O}^{\preceq\ell}, q^{\preceq\ell})\}$
16: **else return** $\{\langle r_i\rangle \hookrightarrow \langle \gamma_{exit}\rangle\ (x' = \mathcal{I}_i[p_i]) \wedge (p'_i = p_i + 1) \wedge rt(\mu \setminus \{x\}, \mathcal{I}^{\preceq\ell}, p^{\preceq\ell} \setminus \{p_i\}, \mathcal{O}^{\preceq\ell}, q^{\preceq\ell})\}$ //$\sigma(\mathcal{I}_i) \preceq \ell$
17: **else if** C is $r_i : output(e, \mathcal{O}_i)$ **then** //Confidential and public outputs
18: **if** $\sigma(\mathcal{O}_i) \succ \ell$ **then return** $\{\langle r_i\rangle \hookrightarrow \langle \gamma_{exit}\rangle\ rt(\mu, \mathcal{I}^{\preceq\ell}, p^{\preceq\ell}, \mathcal{O}^{\preceq\ell}, q^{\preceq\ell})\}$
19: **else** //$\sigma(\mathcal{O}_i) \preceq \ell$
20: **return** $\{\langle r_i\rangle \hookrightarrow \langle out^i_e, \gamma_{exit}\rangle\ (t' = e) \wedge rt(\mathcal{I}^{\preceq\ell}, p^{\preceq\ell}, \mathcal{O}^{\preceq\ell}, q^{\preceq\ell}) \wedge rt_2(\mu), \langle out^i_x\rangle \hookrightarrow \langle \epsilon\rangle\ rt(\mathcal{I}^{\preceq\ell}, p^{\preceq\ell}, \mathcal{O}^{\preceq\ell}, q^{\preceq\ell})\}$
21: **end procedure**

$\langle r_i\rangle \hookrightarrow \langle out^i_e, \gamma_{exit}\rangle$ stands for calling procedure out^i, and $\langle out^i_x\rangle \hookrightarrow \langle \epsilon\rangle$ stands for exiting from procedure out^i, as shown in Algorithm 2. t' is a global variable used to store the value of expression for a later output. For a rule $\langle \gamma_j\rangle \hookrightarrow \langle \mathsf{f}_{\text{entry}}\gamma_k\rangle$, rt_2 derives a logical relation to express the retainment on value of the local variables of the caller of procedure f. The body of procedure out^i is left blank for the complementary pushdown rules of public outputs to the channel O_i. These rules are added later in Subsection 5.2.

The program variables are compacted with respect to finite domains. For the modeling of I/O channels, G is the domain of $\mathcal{I}^{\preceq\ell} \times p^{\preceq\ell} \times \mathcal{O}^{\preceq\ell} \times q^{\preceq\ell}$. L is the domain of μ, which is actually the range of the mapping function of variables. The stack alphabet reduces to the set of reference labels R. Finally the derived pushdown system of program P has a form of $(dom(\mathcal{I}^{\preceq\ell} \times p^{\preceq\ell} \times \mathcal{O}^{\preceq\ell} \times q^{\preceq\ell}), R \times dom(\mu), \text{GenTrans}(P, \gamma_{exit}, \ell))$.

5.2 Model transformation and complementary rules for I/Os

The new self-composition for reachability analysis [11] consists of three phases: (i) the basic self-composition, (ii) the auxiliary initial interleaving assignments, and (iii) the illegal-flow state construction. Self-composition usually involves two runs of a specific program. The program is composed with another copy of itself, which we called *pairing part* of the composition result. According to the information flow properties, the variables of the pairing part should be renamed considering these variables should be compared with the variables of the original program. These two runs of program can execute either consecutively or interleavingly. In this paper, we take a consecutive order for the composition of program.

We use our previous effort on self-composition, i.e. compact self-composition [29], as the basic self-composition. We briefly summarize the new self-composition in Algorithm 3. The procedure CompactSC performs the transformation from pushdown system $\mathcal{P}$ to another pushdown system. TransSC gives the transformation of pushdown rules. ξ is a renaming function on stack symbols and variables. $\mathcal{R}_{x\in V}[\xi(x)/x]$ is a logical relation derived by substituting each variable of $\mathcal{R}$ in V with a renamed companion variable.

In TransSC, we do not construct the pushdown rule for ending the first run of program and entering the second run of program, i.e. $\langle \gamma_{exit}\rangle \hookrightarrow \langle \xi(\gamma_{init})\rangle$ where $\langle \gamma_{exit}\rangle \hookrightarrow \langle \epsilon\rangle$ is the last transition. This pushdown rule is treated as a model of the initial interleaving assignments. This facilitates assigning of the renamed

Algorithm 3. Compact Self-Composition

```
1: procedure CompactSC(P, γ_init, γ_exit, ℓ)                                  //P = (G, Γ × dom(μ), Δ)
2:   reset ← {⟨γ_exit⟩ ↪ ⟨ξ(γ_init)⟩ (⋀_{p_i∈p^{⪯ℓ}} (p'_i = 0) ∧ ⋀_{q_i∈q^{⪯ℓ}} (q'_i = 0) ∧ rt(μ, ξ(μ), I^{⪯ℓ}, O^{⪯ℓ}))}
3:   out ← ∅
4:   for all i such that O_i ∈ O^{⪯ℓ} do          //Generate pushdown rules of output for store-match pattern
5:     out ← out ∪ {⟨out^i_e⟩ ↪ ⟨out^i_x⟩ (O'_i[q_i] = t) ∧ (q'_i = q_i + 1) ∧ rt(I^{⪯ℓ}, p^{⪯ℓ}, O^{⪯ℓ} \ {O_i[q_i]}, q^{⪯ℓ} \ {q_i})}    //store
6:     out ← out ∪ {⟨ξ(out^i_e)⟩ ↪ ⟨ξ(out^i_x)⟩ (O_i[q_i] = t) ∧ (q'_i = q_i + 1) ∧ rt(I^{⪯ℓ}, p^{⪯ℓ}, O^{⪯ℓ}, q^{⪯ℓ} \ {q_i}),
                ⟨ξ(out^i_e)⟩ ↪ ⟨error⟩ (O_i[q_i] ≠ t)}                                          //match
7:   end for
8:   return (G, (Γ ∪ ξ(Γ)) × dom(μ × ξ(μ)), TransSC(Δ) ∪ reset ∪ out)
9: end procedure
10: procedure TransSC(Δ)
11:   Δ' ← ∅
12:   for all (⟨γ_i⟩ ↪ ⟨γ_j⟩ R) ∈ Δ do
13:     Δ' ← Δ' ∪ {⟨γ_i⟩ ↪ ⟨γ_j⟩ R ∧ rt(ξ(μ)), ⟨ξ(γ_i)⟩ ↪ ⟨ξ(γ_j)⟩ R_{x∈V}[ξ(x)/x] ∧ rt(μ)}
14:   for all (⟨γ_i⟩ ↪ ⟨f_entry γ_j⟩ R) ∈ Δ do
15:     Δ' ← Δ' ∪ {⟨γ_i⟩ ↪ ⟨f_entry γ_j⟩ R ∧ rt_2(ξ(μ)), ⟨ξ(γ_i)⟩ ↪ ⟨ξ(f_entry)ξ(γ_j)⟩ R_{x∈V}[ξ(x)/x] ∧ rt_2(μ)}
16:   for all (⟨γ_i⟩ ↪ ⟨ε⟩ R) ∈ Δ do
17:     if γ_i ≠ γ_exit then Δ' ← Δ' ∪ {⟨γ_i⟩ ↪ ⟨ε⟩ R, ⟨ξ(γ_i)⟩ ↪ ⟨ε⟩ R}
18:     else Δ' ← Δ' ∪ {⟨ξ(γ_exit)⟩ ↪ ⟨ξ(γ_exit)⟩ R_{x∈V}[ξ(x)/x] ∧ rt(μ)}              //For ⟨γ_exit⟩ ↪ ⟨ε⟩ R
19:   end for
20:   return Δ'
21: end procedure
```

public variables with the value of the original public variables initially, to set free the precondition on the initial state for the security condition. In order to avoid duplicating the public input channels in $\mathcal{I}^{\preceq\ell}$, we reduce the initial interleaving assignments by reusing the content of public input channels. This is achieved by resetting the anchor indexes in $p^{\preceq\ell}$ to 0 at the beginning of the pairing part of model to reread these channels. Meanwhile, the indexes of public output channels are also reset for the subsequent inequality checks of outputs. The complementary pushdown rule is the *reset* in Algorithm 3.

Leveraging ordinary self-composition [12] on public outputs to the channels in $\mathcal{O}^{\preceq\ell}$ will duplicate these channels and largely increase the state space of abstract model. The pioneering advantage of *store-match* pattern is that it reduces the state space. In order to avoid duplication on public output channels, we should *match* the intentional output of the second run with the corresponding output *stored* in the first run of program. The abstraction of public outputs employs the store-match pattern. We compare the newly derived public output to channel O_i in the second run with the corresponding public output stored by the first run. If they are equal, the new output is discarded and the symbolic execution progresses, otherwise the symbolic execution is directed to an illegal-flow state, i.e. *error* in CompactSC. The complementary pushdown rules for the public outputs to O_i are parameterized with channel identifier i and generated to derive the set *out* in CompactSC. As a result, the body of output left vacuous in GenTrans of Algorithm 2 is complemented with the set *out* in CompactSC.

5.3 Equivalence on declassifiable expressions

In Algorithm 2, we have not demonstrated the abstraction of declassification command. Considering the semantics in Figure 2, the computation of declassification is similar to an ordinary assignment for variable for obtaining the declassifiable expression value. Consequently, GenTrans($r : x := declass(e, r_0), r', \ell$) derives the following pushdown rule

$$\langle r\rangle \hookrightarrow \langle r'\rangle\ (x' = e) \wedge rt(\mu \setminus \{x\}, \mathcal{I}^{\preceq\ell}, p^{\preceq\ell}, \mathcal{O}^{\preceq\ell}, q^{\preceq\ell}).$$

From the definition of R3P and the proof for Theorem 3, we should enforce $\bigwedge_{1\leqslant i<j}(\mathcal{H}(r_i,\ell) = \mathcal{H}(r'_i,\ell) \wedge \bigwedge_{e\in\mathcal{H}(r_i,\ell)}(\mu_i(e) = \mu'_i(e)))$ as preconditions and $\mu_j \sim_\ell \mu'_j (1 \leqslant j \leqslant k)$ as postconditions progressively along

the correlative runs of program. The content of $\mathcal{H}(r_i, \ell)$ is relevant to the security domain ℓ. In order to make R3P enforceable with reachability analysis, we should be able to troubleshoot the following issues.

5.3.1 *Modeling the equivalence on declassifiable expressions*

The equivalence on declassifiable expressions at a specific reference point should be enforced as precondition of the security property. First, each reference point r with $\mathcal{H}(r, \ell) \neq \emptyset$ is detected by prior static collection of the security policy from the code. Each of these reference points corresponds to some stack symbol, e.g., r_s, and there are t pushdown rules in GenTrans(C, γ_{exit}, ℓ) with a form of $\langle r_s \rangle \hookrightarrow \langle r_i \rangle$ $(\mathcal{R}_i)$ where $1 \leqslant i \leqslant t$. Each of these pushdown rules should be modified to $\langle yr_s \rangle \hookrightarrow \langle r_i \rangle$ $(\mathcal{R}_i)$. The store-match actions can then be added between r_s and yr_s as complementary pushdown rules.

Some structures as instrument are required for abstract modeling to ensure the progression of computation of the second run only when the equalities on value of declassifiable expressions are satisfied. Considering this, we should define the sites for storing values of declassifiable expressions at specific reference points. For each security domain ℓ, if there are k reference points $r_1, \ldots, r_k$, with non-empty set of declassifiable expressions, we define $Q_{r_1}, \ldots, Q_{r_k}$ as global lists to store the values of these expressions. The length of each Q_{r_i} is $|\mathcal{H}(r_i, \ell)|$. Accompanied with each of these global lists, we define $\rho_{r_i} : \mathcal{H}(r_i, \ell) \mapsto \mathbb{N}$ to map each declassifiable expression in $\mathcal{H}(r_i, \ell)$ to an index of Q_{r_i}. Finally, we give the complementary pushdown rules for the store-match actions as follows:

$$\begin{aligned}
&Dcl_{sto} \quad \langle r_s \rangle \hookrightarrow \langle yr_s \rangle \ (\textstyle\bigwedge_{e \in \mathcal{H}(r_s,\ell)} Q'_{r_s}[\rho_{r_s}(e)] = e) \wedge rt(Q_{\{r_1,\ldots,r_k\} \setminus \{r_s\}}, \mu, \xi(\mu), \mathcal{I}^{\preceq \ell}, p^{\preceq \ell}, \mathcal{O}^{\preceq \ell}, q^{\preceq \ell}).\\
&Dcl_{mat} \quad \langle xr_s \rangle \hookrightarrow \langle \xi(yr_s) \rangle \ (\textstyle\bigwedge_{e \in \mathcal{H}(r_s,\ell)} Q_{r_s}[\rho_{r_s}(e)] = \xi(e)) \wedge rt(Q_{\{r_1,\ldots,r_k\}}, \mu, \xi(\mu), \mathcal{I}^{\preceq \ell}, p^{\preceq \ell}, \mathcal{O}^{\preceq \ell}, q^{\preceq \ell}),\\
&\qquad\quad\ \ \langle xr_s \rangle \hookrightarrow \langle idle \rangle \ (\textstyle\bigvee_{e \in \mathcal{H}(r_s,\ell)} Q_{r_s}[\rho_{r_s}(e)] \neq \xi(e)).
\end{aligned}$$

The matching phase is modeled from xr_s to $\xi(yr_s)$, but not from $\xi(r_s)$ to $\xi(yr_s)$. The gap from $\xi(r_s)$ to xr_s is left for subsequent usage. The state *idle* only launches transition to itself, therefore the reach of *idle* implies that the state *error* in Algorithm 3 becomes unreachable. This condition implying the violation of the precondition of security property rules out the traces irrelevant to the decision of property.

5.3.2 *Modeling the violation of ℓ-indistinguishability on memory stores*

Apart from the symbolic computation directed by the inequality of outputs to state *error*, the violation of ℓ-indistinguishability on memory stores can also cause illegal flow. Modeling the ℓ-indistinguishability on memory stores at a specific reference point, i.e. $\mu_j \sim_\ell \mu'_j$, is similar to that of store-matches for public outputs except that an output channel O_μ is defined with length $k \cdot |V|$ instead of O_i for the first run of program. The storing phase accompanies with the Dcl_{sto} rule and changes the Dcl_{sto} to:

$$\begin{aligned}
Dcl'_{sto} \quad \langle r_s \rangle \hookrightarrow \langle yr_s \rangle \ &(\textstyle\bigwedge_{x_i \in V, \sigma(x_i) \preceq \ell} O'_\mu[q_\mu + i] = x_i) \wedge (q'_\mu = q_\mu + |V|) \wedge \\
&(\textstyle\bigwedge_{e \in \mathcal{H}(r_s,\ell)} Q'_{r_s}[\rho_{r_s}(e)] = e) \wedge rt(Q_{\{r_1,\ldots,r_k\} \setminus \{r_s\}}, \ldots).
\end{aligned}$$

The matching phase is modeled between $\xi(r_s)$ and xr_s with the following complementary pushdown rules:

$$\begin{aligned}
Mat \quad &\langle \xi(r_s) \rangle \hookrightarrow \langle xr_s \rangle \ (\textstyle\bigwedge_{x_i \in V, \sigma(x_i) \preceq \ell} O_\mu[q_\mu + i] = \xi(x_i)) \wedge (q'_\mu = q_\mu + |V|) \wedge rt(O_\mu, \ldots),\\
&\langle \xi(r_s) \rangle \hookrightarrow \langle error \rangle \ (\textstyle\bigvee_{x_i \in V, \sigma(x_i) \preceq \ell} O_\mu[q_\mu + i] \neq \xi(x_i)).
\end{aligned}$$

The index q_μ should be as per the *reset* rule in Algorithm 3. The final symbolic pushdown system returned by CompactSC of Algorithm 3 will be $(dom(\mathcal{I}^{\preceq \ell} \times p^{\preceq \ell} \times \mathcal{O}^{\preceq \ell} \times q^{\preceq \ell} \times Q_{\{r_1,\ldots,r_k\}} \times O_\mu \times \{q_\mu, t\}), (\Gamma \cup \xi(\Gamma) \cup \{xr_i, yr_i \mid 1 \leqslant i \leqslant k\}) \times dom(\mu \times \xi(\mu)), \text{TransSC}(\Delta) \cup reset \cup out \cup Dcl'_{sto} \cup Dcl_{mat} \cup Mat)$.

Theorem 6 (Correctness)**.** Let $\mathcal{P}^\ell$ be the symoblic pushdown system of program P with respect to security domain ℓ. If for all ℓ in $\mathcal{D}$, the state *error* of $\mathcal{P}^\ell$ is unreachable from any initial state, then P complies with R3P.

Proof. Suppose $ER(i, e)$ is a parameterized command (if $O_i[q_i] \neq e$ then goto *error* else $q'_i = q_i + 1$). We first define three substitution functions. $Sub_1(\ell, P) \equiv P[\mathbf{skip}/output(_, O_i)]_{\sigma(O_i) \succ \ell}$ means substituting

each output to the channels whose security domain is higher than ℓ with command **skip**. $Sub_2(\ell, P) \equiv Sub_1(\ell, P)[x := \bot/input(x, I_j)]_{\sigma(I_j)\succ\ell}$ is defined on a result of Sub_1. For each input from the channels whose security domain is higher than ℓ, Sub_2 substitutes the input with an assignment. The variable x is assigned with an indefinite value. $Sub_3(\ell, P) \equiv Sub_2(\ell, P)[ER(i, e)/output(e, O_i)]_{\sigma(O_i)\preceq\ell}$ substitutes the outputs with instantiated ER commands when the security domain of these output channels are not higher than ℓ. Sub_1 and Sub_2 respectively stand for the abstraction of confidential outputs and confidential inputs, while Sub_3 applies the matching phase of public outputs on the result of Sub_2.

From the perspective of public observers, the semantical induction of the normal execution of P is identical to the semantical induction of the execution of $Sub_1(\ell, P)$, as the confidential outputs have no impact on the subsequent computation. When P violates R3P, i.e. $\exists\ell_0.(P, P) \notin \mathbb{R}^{\ell_0}$, we have $(Sub_1(\ell_0, P), Sub_1(\ell_0, P)) \notin \mathbb{R}^{\ell_0}$. As for the reason, the reachability analysis exhaustively checks traces from any possible initial configuration, the domain of each element in $I_i(\sigma(I_i) \succ \ell)$ is identical to indefinite value $\bot$. $Sub_1(\ell, P)$ and $Sub_2(\ell, P)$ have the same set of traces and $(Sub_2(\ell_0, P), Sub_2(\ell_0, P)) \notin \mathbb{R}^{\ell_0}$. Hence, for $Sub_2(\ell_0, P)$, we have (1) $\mu \sim_{\ell_0} \mu'$, and (2) there exists j such that $1 \leqslant j \leqslant k$ and $(\mathcal{I}_j \sim_{\ell_0} \mathcal{I}'_j) \wedge \bigwedge_{1\leqslant i<j}(\mathcal{H}(r_i, \ell_0) = \mathcal{H}(r'_i, \ell_0) \wedge \bigwedge_{e\in\mathcal{H}(r_i,\ell_0)} \mu_i(e) = \mu'_i(e))$, and (3) at least one of the following conditions holds

(3.a) there exists $x \in V$ such that $\sigma(x) \preceq \ell_0$ and $\mu_j(x) \neq \mu'_j(x)$, or

(3.b) there exists an output channel O_x, $\sigma(O_x) \preceq \ell_0$ and at the point j, we have $q_x \neq q'_x \vee \exists 0 \leqslant t < q_x.O_x[t] \neq O'_x[t]$.

Suppose P is like $C_a; r_j : c_j; C_b$. The trace of $Sub_2(\ell_0, P)$ has a form of $(\mu, \mathcal{I}^{\preceq\ell_0}, \mathcal{O}^{\preceq\ell_0}, p^{\preceq\ell_0}, q^{\preceq\ell_0}, Sub_2(\ell_0, P)) \twoheadrightarrow \ldots \twoheadrightarrow (\mu_j, \mathcal{I}^{\preceq\ell_0}, \mathcal{O}_j^{\preceq\ell_0}, p_j^{\preceq\ell_0}, q_j^{\preceq\ell_0}, Sub_2(\ell_0, r_j : c_j; C_b)) \twoheadrightarrow \ldots \twoheadrightarrow (\mu_k, \mathcal{I}^{\preceq\ell_0}, \mathcal{O}_k^{\preceq\ell_0}, p_k^{\preceq\ell_0}, q_k^{\preceq\ell_0}, \mathbf{skip})$. From the pushdown rule *reset* in Algorithm 3, we know after the *reset*, $(p_k^{\preceq\ell_0}, q_k^{\preceq\ell_0})$ becomes $(p^{\preceq\ell_0}, q^{\preceq\ell_0})$. Because $\bigwedge_{1\leqslant i<j}(\mathcal{H}(r_i, \ell_0) = \mathcal{H}(r'_i, \ell_0) \wedge \bigwedge_{e\in\mathcal{H}(r_i,\ell_0)} \mu_i(e) = \mu'_i(e))$, the trace of $Sub_3(\ell_0, P)$ from $(\xi(\mu), \mathcal{I}^{\preceq\ell_0}, \mathcal{O}_k^{\preceq\ell_0}, p^{\preceq\ell_0}, q^{\preceq\ell_0}, Sub_3(\ell_0, P))$ cannot reach *idle* until r'_j. If there exists O_x such that $\sigma(O_x) \preceq \ell_0$ and $q_x \neq q'_x \vee \exists 0 \leqslant t < q_x.\ O_x[t] \neq O'_x[t]$ at j, then the trace of $Sub_3(\ell_0, P)$ should reach *error* because there exists $ER(x, O'_x[t])$. If there exists $x \in V$ such that $\sigma(x) \preceq \ell_0 \wedge \mu_j(x) \neq \mu'_j(x)$, then there exists $\xi(x_i)$ where $\sigma(\xi(x_i)) \preceq \ell_0$ and $\xi(x_i) \neq O_\mu[q_\mu + i]$ at j, where *error* is reachable according to the second pushdown rule of Mat. Finally, from the contrapositive the theorem is proved.

6 Evaluations

We have extended the parser of Remopla[2)] to adapt the model translation proposed in Section 5 as part of the parser. The back-end model checker we use is Moped[3)]. Moped employs Binary Decision Diagrams[4)] to implement the compact representation of $\mathcal{R}$ for each pushdown rule. The purpose of evaluation has generated three significant outcomes: First, in addition to the explanations in Subsection 4.3, we illustrate the difference between the security properties (i.e. WERP, relaxed noninterference, and R3P) with more examples from [7]. Second, we also compare the preciseness of respective enforcements between our reachability analysis and the type system [7] as well as the safety verification [8]. Third, we evaluate the effect of the length of channels, as well as the number of reference points and declassifiable expressions.

The test cases C_1 to C_7 in Table 3 are from Table 1, and $P'_3, P'_6, P_8, P_9, P'_9$ are from [7]. The row *WERP*, *RNI* and *R3P* respectively record whether the test case satisfies the security property specified by WERP, relaxed noninterference and R3P. On the cases from [7], R3P displays a similar property as WERP. The difference between WERP and R3P has been illustrated by C_4 in Subsection 4.3. According to WERP, the test case P'_6 $(r = \text{ref}_1)$ is insecure because from the definition of *ih* in [7, Section 5.1], the security property requires a final avg = avg′, but the indefinite input makes it a violation. The program $P_8(r = \text{ref}_1)$ is a similar case. The relaxed noninterference is equivalent to the delimited release for imperative languages, and both properties are 2-*safety* property [8]. WERP can be modified to delimi-

2)http://www2.informatik.uni-stuttgart.de/fmi/szs/tools/moped/remopla-intro.pdf.
3)http://www.fmi.uni-stuttgart.de/szs/tools/moped/.
4)http://vlsi.colorado.edu/∼fabio/CUDD/.

Table 3 Comparison on security properties and enforcements

Case	*WERP*	*Type*	*RNI*	*BLAST*	*R3P*	*RA*
C_1	×	×	×	×	×	×
C_2	×	×	–	–	×	×
C_3	✓	✓	✓	✓	✓	✓
C_4	×	×	✓	✓	✓	✓
C_5	✓	✓	✓	✓	✓	✓
C_6	×	×	×	×	×	×
C_7	✓	×	✓	✓	✓	✓
$P_3'(r = \mathrm{ref}_1)$	×	×	×	×	×	×
$P_6'(r = \mathrm{ref}_1)$	×	×	×	×	×	×
$P_6'(r = \mathrm{ref}_{101})$	✓	✓	–	–	✓	✓
$P_8(r = \mathrm{ref}_1)$	×	×	×	×	×	×
$P_8(r = \mathrm{ref}_{101})$	×	×	–	–	×	×
$P_9'(r = \mathrm{ref}_1)$	×	×	×	×	×	×
$P_9(r = \mathrm{ref})$	✓	✓	–	–	✓	✓

ted release by setting the reference points at the initial state [7]. In order to reduce R3P to relaxed noninterference, a similar method is opted by setting the reference points of each declassification at the initial state. Concurrently, the decision of relaxed noninterference is irrelevant to the location of reference points and comparable to WERP and R3P only on the cases with $r = \mathrm{ref}_1$.

Further, the precision of the enforcements has been evaluated. The row *Type* in Table 3 gives the well-typeness of program decided by the type and effect system in [7, Figure 2]. The row *RA* is the result of reachability analysis using the algorithms in Section 5. Here ✓/× means the illegal-flow state *error* is unreachable/reachable. The row *BLAST* is the verification result based on the self-composition given in [8] and the model checker BLAST[5)]. The experimental results show that our mechanism can precisely enforce R3P. Our enforcement is exceedingly generalized compared with the safety verification [8], as our mechanism is available to enforce relaxed noninterference by setting $r = \mathrm{ref}_1$, however, the safety verification cannot enforce R3P or WERP. The reason is that their enforcement has no proposed mechanism for working with the intermediate ℓ-indistinguishability $\mu_j \sim_\ell \mu_j'$. The type system [7, Figure 2] conservatively rejects some value-dependent secure program, e.g., C_7, because using this type system we have $\emptyset \vdash h_1 - h_1$:high but high $\not\leq$ $dom(l)$ =low. On the contrary, our reachability analysis can correctly decide this case to be secure.

We choose more complicated examples, adpcm, jfdcint, nsichneu, statemate, from the Mälardalen WCET benchmarks[6)] to evaluate the effects of (1) the number of declassifiable expressions, (2) the location and number of reference points, and (3) the length of channels. We model these examples with Remopla. The standard I/Os are treated as the I/Os to the channels, and the returned value of the main procedure is treated as public output to the channel. We randomly select the declassified expressions from the global variables. The experimental environment is 2.83 GHz×4 Intel CPU, 4 GB RAM, Linux kernel 2.6.38-8-generic.

The experimental results have been illustrated in Figures 3–6. In each figure, the x-axis defines the length of the channel. In Figures 3 and 5, the y-axis is the analysis time of each test case. In Figures 4 and 6, the y-axis is the peak number of BDD nodes which can be used to measure the memory cost. The notation 1rp_start_$n$exp denotes the model of the program in which there are n expressions to become declassifiable at the start location of the code, i.e. the reference point r_{init}, with nonempty $\mathcal{H}(r_{init}, \ell)$. The notation $n$rp_mid_$n$exp denotes the model of program in which the n different expressions become declassifiable respectively at n different reference points randomly selected in the middle of the code. The time effect and space effect of the number of declassifiable expressions are respectively illustrated in Figures 3 and 4. It is observed that with any length of the channel, declassifying more expressions at

5)http://mtc.epfl.ch/blast.
6)http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

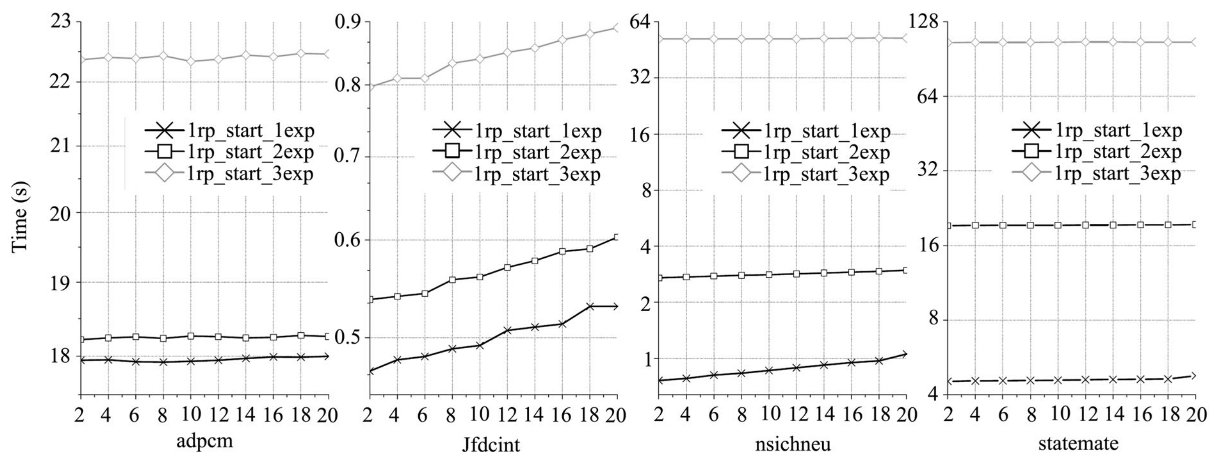

Figure 3 Time effect of the number of declassifiable expressions.

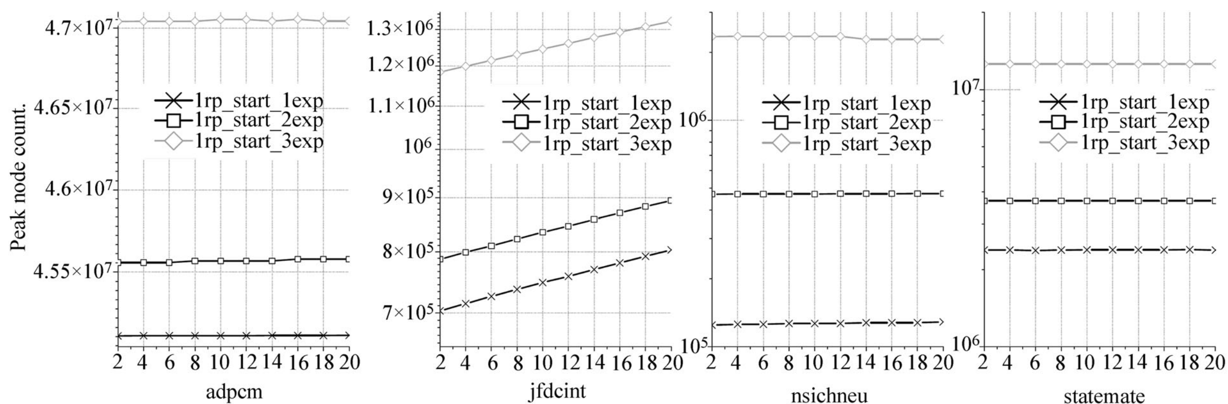

Figure 4 Space effect of the number of declassifiable expressions.

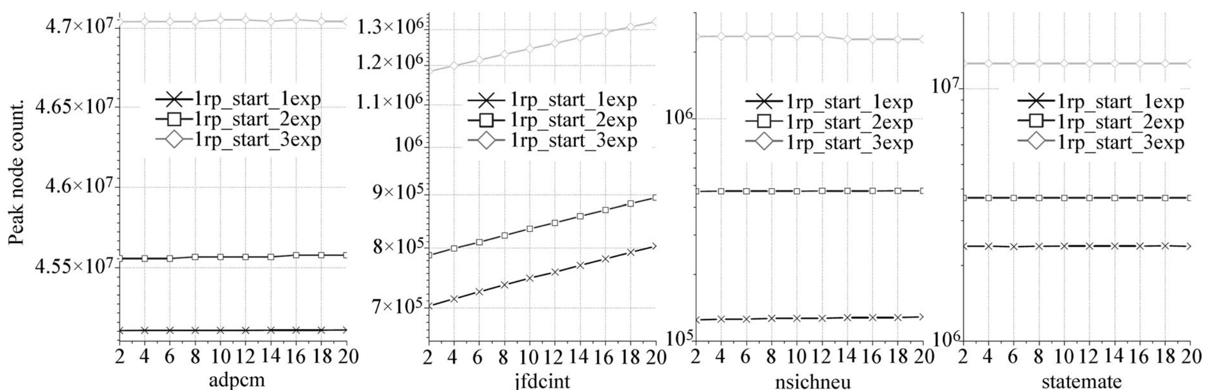

Figure 5 Time effect of the locations and number of reference points.

the same reference point results in more time and space for reachability analysis. This is in conformance with the construction of models in Subsection 5.3 because more declassifiable expressions mean larger $|\mathcal{H}(r_{init}, \ell)|$ and more units in $Q_{r_{init}}$ which lead to an increase in state space and cost of analysis. More declassifiable expressions do not necessarily lead to more cost of analysis, because the positions where to declassify these expressions play a critical role on the cost of analysis. See the case 1rp_start_1exp and 1rp_mid_1exp in Figures 5 and 6. The analysis of the cases with randomly selected locations for declassification can be more efficient (adpcm, jfdcint), less efficient (nsichneu), or with nearly identical cost (statemate) on both time and space. The effect of the number of reference points is also evaluated in Figures 5 and 6. The $(n+1)$rp_mid_$(n+1)$exp model is derived from $n$rp_mid_$n$exp model by declassifying

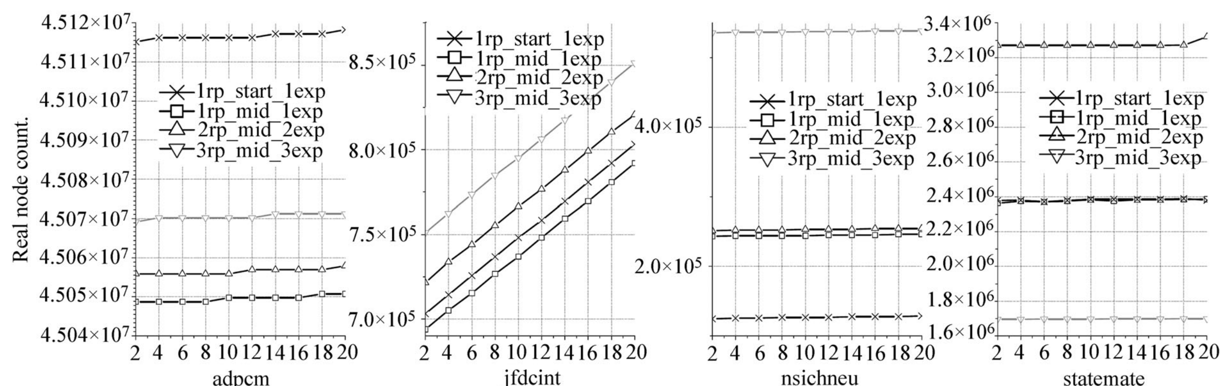

Figure 6 Space effect of locations and number of reference points.

one more expression at a new location. We can see from statemate that adding more reference points to declassify new expressions does not have to increase the cost of analysis. The effects of the length of channels on time and space can be observed from the trend of curves in Figures 3–6. We observe a small time cost increase in most cases along with the growing channel lengths. This is in compliance with the fact that longer channels result in larger state space of a model. The space costs also generally increase along with the growth of channel lengths, but the increased space costs are more insensitive than those of time.

7 Conclusion

We have presented R3P, a what-dimensional information release property for declassification. We have developed an enforcement mechanism using reachability analysis. Aiming to study the enforcement of declassification properties, our work bridges the gap between the flexible declassification policies stressing on different aspects of prudent principles and the existent enforcement techniques based on automated verification. Although our enforcement is developed on pushdown system and BDD-based model checker, our main contributions on store-match pattern are mostly tool-insensitive. Our approach can be adapted to more widely used model checkers with support on reachability analysis and symbolic model checking, e.g. BLAST. Alternative variants of the back-end techniques, e.g. SMT-solver instead of BDD, may also be available considering that our implementation mainly involves the transformation of models. The future work includes channelizing our approach to multi-dimensional properties, as well as scaling up for using on industrial sized code, by implementing a front-end for the translation from C to Remopla.

Acknowledgements

This work was supported by the Key Program of NSFC-Guangdong Union Foundation (Grant No. U1135002), the National Natural Science Foundation of China (Grant No. 61303033), the Major National S&T Program (Grant No. 2011ZX03005-002), the Fundamental Research Funds for the Central Universities (Grant No. JB140309), the Natural Science Basis Research Plan in Shaanxi Province of China (Grant No. 2013JQ8036), and the Aviation Science Foundation of China (Grant No. 2013ZC31003).

References

1 Denning D E. A lattice model of secure information flow. Commun ACM, 1976, 19: 236–243
2 Denning D E, Denning P J. Certification of programs for secure information flow. Commun ACM, 1977, 20: 504–513
3 Goguen J A, Meseguer J. Security policies and security models. In: Proceedings of IEEE Symposium on Security and Privacy, Oakland, California, USA, 1982. 11–20
4 Zdancewic S. Challenges for information-flow security. In: Proceedings of Programming Language Interference and Dependence, 2004
5 Sabelfeld A, Sands D. Declassification: Dimensions and principles. J Comp Secur, 2009, 17: 517–548

6 Sabelfeld A, Myers A C. Language-based information-flow security. IEEE J Select Areas Commun, 2003, 21: 5–19

7 Lux A, Mantel H. Declassification with explicit reference points. In: Backes M, Ning P, eds. Proceedings of the 14th European Symposium on Research in Computer Security. Berlin/Heidelberg: Springer-Verlag, 2009. 69–85

8 Terauchi T, Aiken A. Secure information flow as a safety problem. In: Hankin C, Siveroni I, eds. Proceedings of 12th International Symposium on Static Analysis. Berlin/Heidelberg: Springer-Verlag, 2005. 352–367

9 Sun C, Tang L, Chen Z. A new enforcement on declassification with reachability analysis. In: Proceedings of INFOCOM Workshops, Shanghai, 2011. 1024–1029

10 Lux A, Mantel H. Who can declassify? In: Degano P, Guttman J D, Martinelli F, eds. Proceedings of 5th International Workshop on Formal Aspects in Security and Trust. Berlin/Heidelberg: Springer-Verlag, 2009. 35–49

11 Sun C, Tang L, Chen Z. Secure information flow in Java via reachability analysis of pushdown system. In: Wang J, Chan W K, Kuo F C, eds. Proceedings of the 10th International Conference on Quality Software, Zhangjiajie, 2010. 142–150

12 Barthe G, D'Argenio P R, Rezk T. Secure information flow by self-composition. In: Proceedings of Computer Security Foundations Workshop. Los Alamitos: IEEE, 2004. 100–114

13 Naumann D. From coupling relations to mated invariants for checking information flow. In: Gollmann D, Meier J, Sabelfeld A, eds. Proceedings of the 11th European Symposium on Research in Computer Security. Berlin/Heidelberg: Springer-Verlag, 2006. 279–296

14 Qing S, Shen C. Design of secure operating systems with high security levels. Sci China Inf Sci, 2007, 50: 399–418

15 Bao Y B, Yin L H, Fang B X, et al. A novel logic-based automatic approach to constructing compliant security policies. Sci China Inf Sci, 2012, 55: 149–164

16 Sabelfeld A, Myers A. A model for delimited information release. In: Futatsugi K, Mizoguchi F, Yonezaki N, eds. Software Security - Theoreis and Systems. Berlin/Heidelberg: Springer-Verlag, 2004. 174–191

17 Li P, Zdancewic S. Downgrading policies and relaxed noninterference. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York: ACM, 2005. 158–170

18 Mantel H, Reinhard A. Controlling the what and where of declassification in language-based security. In: De Nicola R, ed. Proceedings of the 16th European Symposium on Programming. Berlin/Heidelberg: Springer-Verlag, 2007. 141–156

19 Adetoye A O, Badii A. A policy model for secure information flow. In: Degano P, Viganò L, eds. Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security. Berlin/Heidelberg: Springer-Verlag, 2009. 1–17

20 Cohen E S. Information transmission in sequential programs. Found Secure Comp, 1978, 297–335

21 Askarov A, Sabelfeld A. Localized delimited release: Combining the what and where dimensions of information release. In: Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security. New York: ACM, 2007. 53–60

22 Lux A, Mantel H, Perner M. Scheduler-independent declassification. In: Gibbons J, Nogueira P, eds. Mathematics of Program Construction. Berlin/Heidelberg: Springer-Verlag, 2012. 25–47

23 Myers A C, Liskov B. A decentralized model for information flow control. In: Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles. New York: ACM, 1997. 129–142

24 Zhou C H, Liu Z F, Wu H L, et al. Symbolic algorithmic verification of intransitive generalized noninterference. Sci China Inf Sci, 2012, 55: 1650–1665

25 Volpano D M, Irvine C E, Smith G. A sound type system for secure flow analysis. J Comp Secur, 1996, 4: 167–188

26 Schwoon S. Model checking pushdown systems. Dissertation for Ph.D. Degree. Munich: Technical University of Munich, 2002

27 Sun C, Zhai E N, Chen Z, et al. A multi-compositional enforcement on information flow security. In: Qing S H, Susilo W, Wang G L, et al., eds. Information and Communications Security. Berlin/Heidelberg: Springer-Verlag, 2011. 345–359

28 Reps T W, Schwoon S, Jha S, et al. Weighted pushdown systems and their application to interprocedural dataflow analysis. Sci Comput Program, 2005, 58: 206–263

29 Sun C, Tang L, Chen Z. Secure information flow by model checking pushdown system. In: Proceedings of the 2009 Symposia and Workshops on Ubiquitous, Autonomic and Trusted Computing, Brisbane, Australia, 2009. 586–591